Conference Paper

Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model

Author(s):
Heußner, Alexander; Poskitt, Christopher M.; Corrodi, Claudio; Morandi, Benjamin

Publication Date:
2015

Permanent Link:

https://doi.org/10.3929/ethz-a-010415005 →

Originally published in:

Electronic Proceedings in Theoretical Computer Science 181, http://doi.org/10.4204/EPTCS.181 →

Towards Practical Graph-Based Verification for an Object-Oriented Concurrency Model

Alexander Heußner
University of Bamberg, Germany

Christopher M. Poskitt Claudio Corrodi
Benjamin Morandi
Department of Computer Science
ETH Zürich, Switzerland

To harness the power of multi-core and distributed platforms, and to make the development of concurrent software more accessible to software engineers, different object-oriented concurrency models such as SCOOP have been proposed. Despite the practical importance of analysing SCOOP programs, there are currently no general verification approaches that operate directly on program code without additional annotations. One reason for this is the multitude of partially conflicting semantic formalisations for SCOOP (either in theory or by-implementation). Here, we propose a simple graph transformation system (GTS) based run-time semantics for SCOOP that grasps the most common features of all known semantics of the language. This run-time model is implemented in the state-of-the-art GTS tool GROOVE, which allows us to simulate, analyse, and verify a subset of SCOOP programs with respect to deadlocks and other behavioural properties. Besides proposing the first approach to verify SCOOP programs by automatic translation to GTS, we also highlight our experiences of applying GTS (and especially GROOVE) for specifying semantics in the form of a run-time model, which should be transferable to GTS models for other concurrent languages and libraries.

1 Introduction

Background Multi-core and distributed architectures are becoming increasingly ubiquitous, as the focus for delivering computing performance shifts from CPU clock speeds—now reaching their natural limits—to concurrency. Harnessing this power, however, requires a fundamentally different approach to writing software; developers must program with concurrency, asynchronicity, and parallelism in mind. Traditionally this has been achieved through threads, synchronising via low-level constructs like locks and semaphores. This approach, while still pervasive, is difficult to master and notoriously error prone; deadlocks, data races, and other concurrency faults are all-too-easy to introduce, yet are challenging to detect and debug. In an effort to alleviate this task for programmers, a number of high-level libraries and languages have been proposed that provide simpler-to-use models of concurrency. Examples include Grand Central Dispatch [11] and SCOOP [20], both of which support asynchronous concurrent programming through abstractions that are safer and simpler to grasp than threads. The concurrency mechanisms of SCOOP, for example, exclude data races by design. Despite such abstractions, programs may still exhibit rich, complex behaviours that are difficult to fully comprehend through testing alone. There is a pressing need for formal models of these systems to facilitate reasoning, comparisons, and understanding, as well as to bring them within reach of current verification tools and techniques.

Initial Problem The intricate features of these libraries and languages—including locking, waiting queues, asynchronous remote calls, and dynamic and automatic thread generation—lead to formal models with verification decision questions (e.g. deadlock detection and the verification of temporal, behavioural properties) that are undecidable. Existing approaches to tackle this theoretical challenge fall

Submitted to:
GaM 2015

mainly into two categories: verification algorithms working on restrictions to simple approximations, e.g. by extended automata models or Petri nets [12, 2], or semi-algorithmic approaches on models that try to cover the original features as faithfully as possible, e.g. by bounded model checking [7].

In the context of SCOOP—a high-level object-oriented concurrency model, implemented as an extension to Eiffel—there are currently no analysis or verification approaches that work directly on a program's source code without additional annotations. Recent first steps into the analysis and prevention of deadlocks in SCOOP are either based on checking Coffman's deadlock conditions on an abstract semantic level [5], or require code to be annotated with locking orders [28]. In earlier work [4], SCOOP programs were translated by hand to models in the process algebra CSP for e.g. deadlock analysis; but these models were too large for the leading CSP tools to cope with, and required a new tool to be custom-built for the purpose (which is no longer maintained today). No further verification approaches for behavioural properties, e.g. specified in some temporal logic, exist yet.

In addition, these concurrent libraries and languages often have semantics that are not fully formally specified, or are associated with multiple semantics—whether existing as formal specifications or implicitly, by implementation. The choice of the "right" semantic formalisation, however, is a substantial prerequisite for the analysis and verification of a program's source code. SCOOP, for example, has at least four established, different semantic formalisations [29, 19, 4, 21]. This "semantic plurality" is an additional source of complication for verification approaches, such as the one we propose in this paper.

Our Approach As a first step, we develop—from the core of the language up—a formal model permitting the simulation and verification of SCOOP programs. The rich semantic features of SCOOP regarding concurrency, (basic) object-orientation, and especially asynchronicity are grasped with the help of graph transformation systems (GTS) that are parameterised by different underlying semantic variants. We also supply a compiler to automate the task of generating input graphs from SCOOP source code. These are then analysed with the help of GROOVE, a state-of-the-art GTS tool, which already includes basic model checking algorithms for GTS.

Contribution & (Closely) Related Work The contribution of the paper is thus manifold: first, we provide a formal GTS-based model that covers SCOOP's basic features and can be seen as a new, additional operational semantics for the language. Second, this GTS-model can also be seen as a new general runtime environment for analysing and verifying object-oriented concurrent programs that share SCOOP's main features, including approximations of SCOOP. Third, the given analysis approach serves as a first step towards a general framework for verifying concurrent asynchronous programs by also highlighting modelling best practices, which can be transferred to the analysis and verification of other libraries, e.g. Grand Central Dispatch, in a similar way. Combining all these aspects, we provide, to our knowledge, the first approach for verifying a subset of SCOOP programs on the code level with respect to behavioural specifications—including deadlock freedom. Only the advanced typing mechanisms and some Eiffel-specific features of SCOOP are currently out of reach for our automatic verification approach.

For the broader verification community, this paper demonstrates how a GTS-based semantics and tool can be effectively used to model, simulate, and facilitate verification for a concurrent programming language that abstracts away from threads and has a "frequently evolving" run-time. For the graph modelling community, this paper presents our experiences of applying a state-of-the-art GTS tool to a non-trivial and practical modelling and verification problem.

The two closest related works are [4] and [19], which both share our first step of providing a new operational semantics for SCOOP. Whereas the former formalises the semantics with the help of a pro-

cess algebraic model in CSP, the latter defines a semantics based on rewriting logic in Maude. Relying on "classical" process algebra, the expression of real asynchronicity between concurrent threads and asynchronous remote method calls are not fully supported by the CSP model—contrary to the model we propose. The comprehensive Maude formalisation is currently seen by the community as the gold standard for SCOOP and coined our understanding of SCOOP's semantics; our model, in contrast, focuses more on the core asynchronous and concurrent features of SCOOP, but can be extended to capture the advanced language features inherited from Eiffel (cf. later comparison in Section 5 for details). Both the CSP and Maude models were used successfully to resolve ambiguities in the original, informal descriptions of SCOOP's semantics, but are insufficient for general verification tasks. Directly harvesting, for example, the more expressive and complete Maude implementation for deadlock analysis does not scale on even toy examples like the Dining Philosophers program (presented later).

Plan of the Paper After introducing SCOOP's main concurrency features (Section 2), we present a formal model which for the sake of simplicity, ignores "local" object-orientation and corresponds to a subset of SCOOP that we will call CoreSCOOP (Section 3). We show how to render CoreSCOOP programs as GTS models (Section 4). Afterwards, we describe how we extended our GTS model for SCOOP to include full object-orientation, and present a workflow for translating SCOOP programs into GROOVE models (Section 5). The latter then allows us to verify programs written in SCOOP with the general algorithms already implemented in GROOVE (Section 6). We conclude with a comparison to related work on GTS-based verification of concurrent object-oriented systems (other references to related work are stated in the corresponding sections) and provide an outlook on our current research.

2 SCOOP: A Concurrent Asynchronous OO Model

SCOOP *Simple Concurrent Object-Oriented Programming* (SCOOP) [20, 19] is a concurrent, asynchronous, and object-oriented programming model that—with its intricate semantics—provided the motivation and challenge for the work in this paper. The most thorough implementation of the model is as an extension to Eiffel, but it has also been explored within the context of Java [27]; we shall focus on the former, and take "SCOOP" in the following to be a synonym for both the model and this principal implementation.

In SCOOP, every object is *handled* by a *processor*, a concurrent thread of control with the exclusive right to call methods on the objects it handles. In this context, object references may point to objects handled by the same processor (*non-separate* objects), or to objects handled by other ones (*separate* objects). Given an object reference `x` and a method `m` that is a *command* (i.e. does not return a result), a method call `x.m` is executed synchronously if `x` is non-separate. If `x` is separate, then the handler of `x` is sent a request to execute the method asynchronously. This latter case is the main source of concurrency in SCOOP programs, which is based essentially on message passing between processors.

The possibility of an object having a different handler is captured in the type system by the keyword `separate`. In order to prevent data races, calls to a separate object `x` are only allowed if the current object's processor holds a lock on the handler of `x`. The programmer does not manage these locking requirements explicitly, but rather expresses them implicitly in the formal argument lists of methods: if the arguments of a method contain separate objects, then the objects' handlers will all be locked (simultaneously, atomically, and automatically—at least conceptually) before the method is executed, and released when it is finished.

Listing 1: Snippet of the PHILOSOPHER class from a Dining Philosophers solution in SCOOP [26]

```
1  live -- Each Philosopher eats times_to_eat times
2    do
3      from until
4        times_to_eat < 1
5      loop
6        print ("Philosopher " + Current.id.out  + " waiting for forks.%N")
7        eat (left_fork, right_fork)
8        print ("Philosopher " + Current.id.out  + " has eaten.%N")
9        times_to_eat := times_to_eat - 1
10     end
11   end
12 eat (left, right: separate FORK) -- Eat, having acquired left and right forks
13   do
14     -- Here, eating takes place
15   end
16 left_fork, right_fork: separate FORK -- References to forks used for eating
```

Dining Philosophers Example A simple example highlighting the intricacies and expressiveness of SCOOP is an implementation of the Dining Philosophers problem: a number of philosophers sit at a round table that provides only single forks between adjacent pairs, and these philosophers must concurrently and correctly alternate between eating and thinking. The caveat of course is that a philosopher may only eat if they hold both the fork to their left and the fork to their right, and algorithms must "pick up" the forks in such a way that prevents a cyclic deadlock from arising. Consider Listing 1, which contains an excerpt from the `PHILOSOPHER` class of a well-known SCOOP solution (available at [26]). Each philosopher and fork object is handled by its own processor. Upon creation, each philosopher is "launched" by calling the (argumentless) `live` method, causing them to concurrently begin the process of eating and thinking. To eat, a philosopher calls the `eat` method, passing the separate object references for the two forks. Eating does not commence until the handlers of these forks are locked by the philosopher's handler; conceptually, this occurs simultaneously, avoiding the possibility of deadlock from e.g. every philosopher locking their left forks only and then waiting indefinitely on their right ones.

Concurrency, Asynchronicity, and Locking SCOOP has a number of other features and behaviours detailed more thoroughly in [19]; here, we briefly describe only queries and contracts. First, given a separate object reference `x` and a method `m` that is a *query* (i.e. returns a result), a call `q = x.m` is always executed synchronously; if `x` has a separate handler, then the current object's handler simply waits for the result to be returned. Secondly, SCOOP maintains and extends the Eiffel tradition of annotating methods with preconditions (keyword `require`) and postconditions (`ensure`). In the sequential setting, these are (optionally) checked before and after every execution of the method. In the concurrent setting, however, preconditions are instead interpreted as *wait conditions* that must be synchronised on. Conceptually, the execution of a routine is delayed until simultaneously the precondition is satisfied and the handlers of the formal arguments are locked. This allows the programmer to express synchronisation conditions at a high level of abstraction.

These concepts require execution-time support from an effective run-time environment. The current run-time [19] associates each processor with a lock and *request queue*. A method call on a separate object is enqueued on its handler's request queue, which is processed in FIFO order. The call may only be enqueued if the lock on the handler is held. The run-time is responsible for correctly synchronising on wait conditions and locking the handlers of formal arguments at the beginning of methods, as well as releasing them at the end. The run-time must balance these design needs with the need to permit

a reasonable level of performance (e.g. by reducing resource contentions). As such, and as a major challenge for our work, the "official" run-time is frequently evolving, and several alternatives have been proposed and/or developed, e.g. [20, 19, 29].

3 A GTS-based Model of CoreSCOOP

Our first step is to present a run-time model for the core behaviours of SCOOP, i.e. remote method calls, FIFO queues, and locking. This model, named *Concurrent Processor Model* (CPM), strips away the object-oriented features of SCOOP, grasping only a subset of the language and focusing on processors equipped with simple data. This allows us to: *(i)* highlight the fundamental peculiarities of SCOOP as model of concurrency in a more fine-grained formal setting, and *(ii)* present the basic building blocks of our approach in more detail, as we extend CPM to include full object-orientation in Section 5.

From CoreSCOOP to CPM Stripping "local" object-oriented features from SCOOP (e.g. self-calls, non-separate calls) and focusing on *remote* synchronous and asynchronous method calls (i.e. queries and commands) via FIFO queues, as well as locking in a concurrent setting, leads to a subset of the SCOOP language we call CoreSCOOP in the following. We formalise the run-time model for CoreSCOOP by the *Concurrent Processor Model* (CPM). CPM is represented by a graph transformation system in which configurations are given by directed graphs conforming to the type graph of Figure 1. Note that the type graph uses a UML-like notation with type attributes and constraints.

The semantic model of CoreSCOOP is inspired by the current "standard" formalisation of SCOOP's semantics in [19]. It consists of a set of *processors* that run concurrently. Each processor is the *handler* of data in local memory, which is represented as a mapping from variable names $(x_1, \ldots x_m)$ to integers. There is *no global shared memory*, only processor local memory, and this memory can only be accessed by or via its processor. Processors sequentially handle method calls via incoming *requests* that are related to a control-flow graph encoding of the underlying CoreSCOOP methods. Thus, a running processor that handles a *current request* is in a *current state* belonging to this *request method type's* control flow graph. Incoming requests are stored by each processor in a FIFO *queue* before being locally executed. Each processor has a finite set of known neighbour processors, i.e. those accessible for synchronous or asynchronous remote calls, which are stored by reference (the variables $r_1, \ldots, r_n$). Processors can dynamically generate new processors (and assign these directly to local reference variables). Each remote call and its context, i.e. the call's parameters, which consist of integer values (i.e. $p_1, \ldots p_k$) and processor references $(r_1, \ldots, r_l)$, is stored as a *request*. Requests implement "value passing", e.g. requests can pass references to newly generated processors. The return value for queries, i.e. synchronous remote calls, is stored in a special local variable accessible to the caller (variable `result`). In CPM, there are neither local calls, i.e. calls to oneself, nor local recursion.

CPM includes explicit locking between processors, i.e. each processor can be locked by at most one (distinct) processor. CoreSCOOP's implicit locking can thus be translated to CPM's explicit locking, however at a different level of granularity. In general, CPM is able to *simulate* the execution of programs written in CoreSCOOP, which is not possible the other way round due to this different level of atomicity in both models. A processor handling a request "walks" the corresponding part of the control-flow graph by updating the current state according to the actions' semantics, given as graph transformation rules (see Section 4 for example configurations and a rule for commands). Handling of queues, local scheduling of each processor (i.e. terminating the currently handled request and advancing to the next in the waiting queue), instantiation of parameters, etc., is handled by global *scheduling rules*, which also assure that

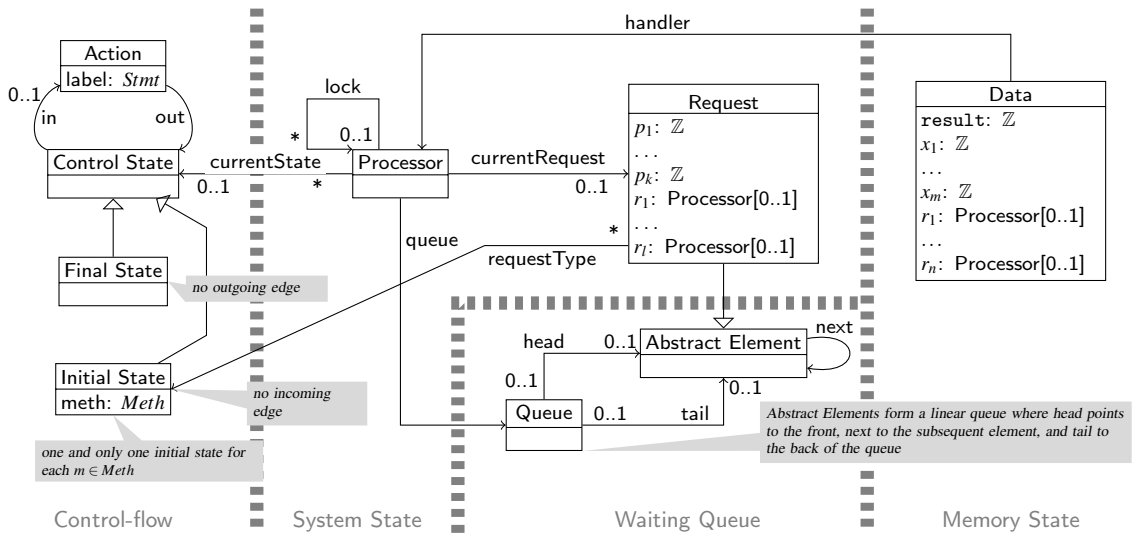

Figure 1: Type graph for CPM's configurations as class diagram with constraints, where we assume disjoint finite sets of variable names ($p_1, \ldots, x_1 \ldots$), and reference names ($r_1, \ldots$); let $Meth$ be a finite set of method names, and $\mathbb{Z}$ be the set of integer numbers, $\mathbb{B}$ of Booleans; the cardinality of association edges is "1" if not noted otherwise; the different regions highlight the different modules of the model

each local processor advances as far as possible. Note that the walking of the control-flow graph and the scheduler are generic, i.e. represented by a set of GTS rules independent of the SCOOP program. We refer to [26] for the full formal model directly represented as GTS (that can be browsed and simulated with GROOVE).

Modularity of CPM Configurations of CPM, i.e. global states of the system, can be partitioned into four main parts, which are also visible in Figure 1: *(i)* a representation of the underlying SCOOP program in the form of a control-flow graph; *(ii)* a system of concurrently running processors, each one possibly handling a request; these processors represent the *system state*; *(iii)* a waiting queue for each processor that stores pending requests; *(iv)* local *memory state* for each processor. The control flow component can be derived directly from the original CoreSCOOP program's control flow graph (at a pre-compilation step) and its structure does not dynamically change, contrary to the state of the run-time environment (processors, queues, data). This partitioning is also mirrored by the GTS's underlying rules that treat the walking of the control flow graph separately from the queue's policies, the management of the memory, and global scheduling.

The fixed simple interfaces (in form of the model components' loose coupling due to the typed associations queue, handler, currentState, and requestType) between these modules allow us to *plug in different behaviours* for each module, e.g. different queueing semantics. Thus we can either adapt different existing SCOOP semantics (e.g. FIFO queues versus queues of queues [29]) or directly apply abstraction mechanisms in the context of verification (e.g. a counting abstraction of the queue's content, or predicate abstraction for the data) by small modular changes to the underlying rules.

Furthermore, the global abstract scheduling rules can be parameterised in this way, e.g. to include different kinds of garbage collection in the global scheduler or different rule prioritisations that keep the state space small, such as always preferring to terminate processors that are currently in a final state.

4 Simulating CPM in GROOVE

We realised CPM—our run-time model for CoreSCOOP—in GROOVE, an established tool for simulating and analysing GTS-based semantics. This section describes how we approached and achieved this task. First, we justify our choice of GROOVE, and then show (by example) how CPM configurations, rules, and rule applications are represented in the tool. Finally, we discuss the issue of CPM's soundness.

The GTS Tool GROOVE We chose *GRaphs for Object-Oriented VErification* (GROOVE) [14, 13] as our platform to implement and analyse the CPM models. Most existing GTS tools are in theory expressive enough to cover CPM. GROOVE however was already applied for the analysis of (non-concurrent) object-oriented programs in Java [24]. Furthermore, GROOVE contains a (finite-state) model checker that has proven sufficient for the analysis and verification of dynamic state systems [7, 18]. As reported in [31], GROOVE can typically handle systems with up to 4 million states, which should leave enough room for our first experiments. Finally, GROOVE convinced us with a gentle learning curve, its ease of adaption and extension to our needs, as well as its active development community.

Representing CPM Configurations in GROOVE CPM configurations are represented in GROOVE quite straightforwardly, with control-flow, system state, waiting queue(s), and memory state (as in the type graph of Figure 1) all encoded in the same graph.

Control-flow is rendered as static transition systems in the graph. These comprise State nodes, where entry points are labelled with *init* and a `method` name, and exit points with *final*. A transition between two State nodes is encoded as a pair of edges (`in` and `out`) and an Action node labelled with a CPM action (e.g. *command*, *query*, *lock*, *assign*). Encoding actions as nodes—as opposed to labelled edges between states—facilitates a clean way of modelling *action parameters*. Simple action parameters, such as a variable to assign to, are encoded as attributes of Action nodes; compound action parameters on the other hand, such as a Boolean expression to be evaluated, are modelled as abstract syntax trees incident to Action nodes. Furthermore, for actions that trigger methods on other processors (i.e. commands, queries), an arbitrary number of *method parameter* nodes (which represent data to be instantiated and available for the duration of a method) can be attached to the corresponding Action nodes. These encode, via attributes, the parameter name, as well as the integer or reference variable to instantiate and pass as the method parameter.

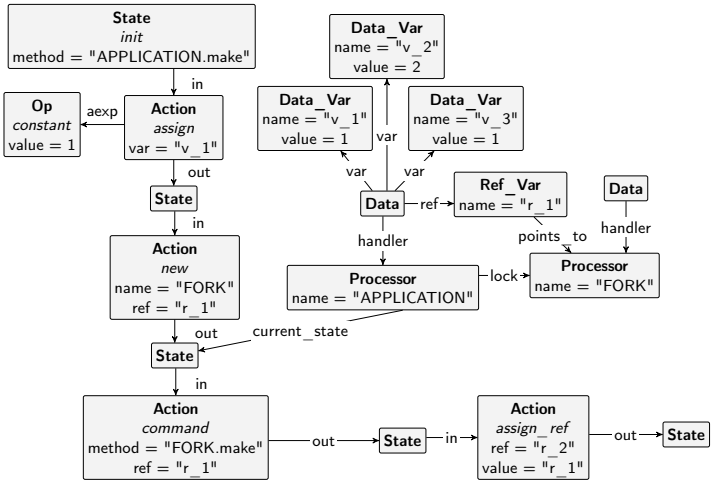

Figure 2: A CPM configuration rendered in GROOVE

System state, waiting queue(s), and memory state are rendered as dynamic parts of the graph. Each Processor node handles a Data node. Data nodes are incident to Data_Var and Ref_Var nodes, which respectively store the handling Processor's integer variables (via attributes) and reference variables (via

edges). Processors may hold locks on other Processors (represented by `lock` edges), and may be in a control State (represented by a `current_state` edge). Furthermore, they have waiting queues of requests to be executed, represented as "linked lists" of Queue_Item nodes over `next`-labelled edges. Each such node is labelled with the `method` (i.e. the particular transition system) to be executed, and is attached to nodes that store the values of any method parameters expected.

An example of a CPM configuration in GROOVE is given in Figure 2. This shows a subgraph of a configuration that can be reached in the CPM encoding of the full Dining Philosophers SCOOP program (see [26]). There are two Processors in this configuration, but only one (APPLICATION) is currently executing a method (`current_state`). This Processor has both a reference to (`points_to`) and a `lock` on the FORK processor (i.e. it has the exclusive right to send requests). Neither Processor is storing any method parameters, and their waiting queues are empty.

Simulating CPM Actions in GROOVE The semantics of CPM is simulated in GROOVE by two sets of graph transformation rules: action rules, and scheduling rules.

Action rules facilitate the firing of transitions in the control-flow part of the graph, and the corresponding updates to the system and memory state. They model the basic behaviours of SCOOP processors: variable assignment, condition evaluation, processor creation, asynchronous commands, synchronous queries, and simultaneously (un)locking multiple processors. An action rule can be applied to a CPM configuration when: *(i)* a processor is in a control-flow state incident to a correspondingly-labelled action; and *(ii)* the prerequisites of the action are satisfied, e.g. every processor targeted by a lock action is available to be locked. Action rules are atomic, in that the firing of a transition occurs in a single, indivisible step (e.g. locking multiple processors occurs instantly, as it appears to SCOOP programmers). This is achieved by extensive use of GROOVE's powerful matching constructs such as *universal quantification*, which allows for a single rule to handle arbitrarily many instances of particular sub-structures (e.g. arbitrary numbers of method parameters). Furthermore, action rules are assigned the same (and lowest) priority in GROOVE, meaning that non-determinism (and thus interleaving) is modelled at the level of atomic processor actions, as opposed to partial evaluations (thereby mitigating one source of state space explosion).

Scheduling rules handle queues, local scheduling of each processor (e.g. advancing to the next request), and any local pre- or post-processing required for action rules; more generally, they advance processors as much as possible in-between actions. While action rules necessarily model non-determinism—different interleavings model different orders in which requests are enqueued, and thus potentially different program outcomes—scheduling rules avoid it as much as possible, since the steps between actions are *local* to processors. This is achieved by rule priorities in GROOVE. In particular, all scheduling rules have higher priorities than action rules, meaning that all local scheduling is simulated before exploring the non-determinism of actions. Furthermore, no two scheduling rules have the same priority, ensuring that their execution is as deterministic as possible to reduce the number of states to explore. Assigning such fine-grained rule priorities did, however, require some care. It is ultimately unimportant, for example, whether a constant or variable in an expression is evaluated first, so we arbitrarily fixed the priority of the scheduling rule for constants to be higher. On the other hand, if we had assigned the scheduling rule for terminating requests (i.e. in final states) to be of higher priority than scheduling rules that perform post-processing immediately after actions (e.g. "resetting" an evaluated expression after assigning it), then a fault would have been introduced into the model.

Let us take a closer look at the action rule for commands, depicted in Figure 3 using GROOVE's colour coding. Recall that in SCOOP (and thus in CPM), a command is an asynchronous remote method

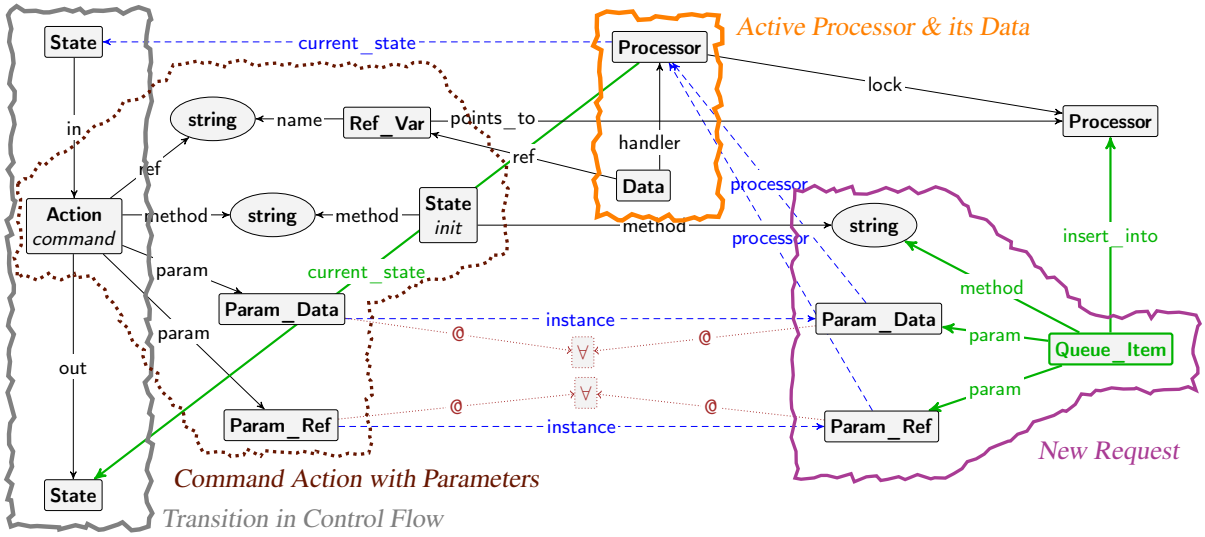

Figure 3: Example action rule for commands in GROOVE (GROOVE's rule colour coding: dashed blue elements only exist on the left-hand side of rule (thus will be deleted), bold green elements on the right-hand side (thus will be generated), black ones persist)

call on a concurrently running processor that is locked by the current processor. The starting point of the rule is an active processor that is currently in a state (current_state) that could fire a command action. The command action is given by the following: *(i)* a pointer (points_to) from a reference variable (Ref_Var) to the target Processor; *(ii)* the name of the method to call, via an attribute of the Action node; and *(iii)* some number of instantiated method parameters (Param_Data, Param_Ref). The firing of the rule advances the processor's current state (current_state) and at the same time generates a new request (as Queue_Item) equipped with the method name and all (via "∀") instantiated parameters. This is then later inserted into the target Processor's waiting queue via a separate, highly prioritised scheduling rule. For illustration, the result of applying the action rule to Figure 2 is given in Figure 4. Observe that the transition has been fired, that the resulting Queue_Item contains the same method name as the Action node, and that it is waiting to be enqueued by the Processor that the reference variable r_1 points to. The current prototype of CPM is available at [26]. At present, it comprises 19 action rules and 34 scheduling rules.

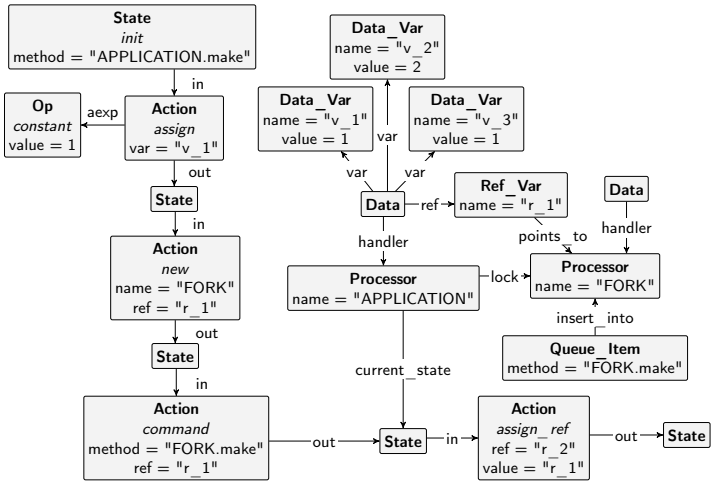

Figure 4: Effect of the command rule on Figure 2

Soundness Formally establishing the soundness (or "faithfulness") of CPM is an important step, which would rely on either the existence of a definitive language semantics, or an in-depth comparison with

one of the proposed semantic approaches to SCOOP. Since the former is lacking, we have attempted to be faithful to the comprehensive operational semantics proposed by Morandi et al. [19] which is "executable" in Maude. To prove soundness with respect to this semantics would however require a more stringent formal representation of CPM, e.g. as graph programs [22], and corresponding semantic proofs (e.g. equivalence/(bi-)simulation). We are addressing this as ongoing work, but it is beyond the scope of this first, proof-of-concept paper.

In the meantime, we took a more lightweight approach to gain confidence in the soundness of CPM, through example-driven testing and an expert review. For the former, we looked at the SCOOP examples supplied with the EiffelStudio IDE, which demonstrate idiomatic usage of SCOOP's concurrency mechanisms to solve a number of classical synchronisation problems. We focused on two programs in particular—Dining Philosophers and Single-Element Producer Consumer—which only ever create one object per processor, and thus were CoreSCOOP programs that straightforwardly map to CPM actions (recall that CPM does not model the notion of multiple local objects). With these programs, we then tested the faithfulness of CPM by: *(i)* visualising and manually inspecting program executions in the GROOVE simulator; *(ii)* exploring the state space for abnormal states (e.g. unsatisfied action pre-requisites, such as the absence of a lock edge for a command) using the LTL model checker (see Section 6); and *(iii)* comparing the effects of action rules against the informal and formal descriptions of SCOOP in [19]. In addition to testing, we also held a one-day "expert review" with B. Morandi and other SCOOP researchers from ETH, during which we demonstrated and discussed the CPM rules in detail with the goal to ensure that the rules fully corresponded to their understanding of SCOOP/CoreSCOOP.

5 Towards Full-Fledged SCOOP, Approximations, and Translations

In this section, we look beyond CPM and CoreSCOOP to consider three ongoing extensions to the work. First, we describe our effort to extend the model with full object-orientation, and thus make it expressive enough for a wider class of SCOOP programs. Second, we discuss how CPM can be used as a basis for exploring alternative SCOOP semantics and model approximations. Finally, we report on a prototype tool for automatically generating GROOVE input from SCOOP (and thus also CoreSCOOP) programs.

From CPM to CPM+OO CPM allowed us to "boil down" SCOOP to the core of its asynchronous, concurrent behaviour, and study it in a formal setting without the full complexity of object-orientation. Our aim however is practical verification, and in practice, SCOOP programmers extensively use objects: ultimately we need to support them. The benefits of a simpler formal model aside, one might wonder why we did not start with full object-orientation from the outset if practical verification was always in mind. This is because it allowed a separation of concerns: we could first isolate and model concurrency-via-processors in a clean, simple setting, and then separately extend it with the object-oriented and Eiffel-specific languages features that are not core to the concurrency model. Our modelling approach has essentially been to identify this core, formalise it, then gradually add the missing details and behaviours.

We are extending CPM to *CPM with Object-Orientation* (CPM+OO), a richer run-time model capable of expressing and simulating SCOOP programs with multiple objects per processor and non-separate method calls (i.e. targetting local objects). The present version of CPM+OO is the result of the following process: *(i)* replacing simple data in the CPM type graph with objects; *(ii)* updating the rules that then no longer conform, in consultation with the semantics of [19]; and *(iii)* testing (including regression testing for CoreSCOOP programs).

Our first goal was to support all of the *existing* actions of CPM, but with data organised into objects.

We began by updating the type graph, replacing simple data with object nodes connected to attributes; attributes being integers (as before) or references to other *objects* (not processors). The advantage in changing the type graph first is the instant feedback from GROOVE, which highlights the rules that no longer conform and thus need updating (i.e. every rule that processed or extracted data). In general, these were not radical updates: the core behaviour captured in CPM remained the same, and the semantics of actions did not fundamentally change. What had to be updated was the structure of data that sat on top of this core, as well as remote calls to processors which became remote calls to objects. In other words, the question we were asking at each step was "how do we correctly embed objects into the semantics of this action" and emphatically *not* "how do we model this asynchronous behaviour for objects".

With CPM "objectified", we could turn to modelling behaviours only possible with data organised into objects, most notably, non-separate calls (calls to objects on the same processor). There is of course no reason to acquire locks in the non-separate case, and the processor simply executes the method immediately. To model this, we had to first model the call stack, also allowing us to capture recursion and local variables—important, practical details, but ultimately on top of (and not crucial to) the concurrency at the core. The present prototype of CPM+OO, available to download from [26], includes all of the features discussed, as well as arbitrary names for attributes (e.g. `buffer` instead of `r_1`), separate queries in expressions, reference expressions, and (optional) postcondition checking.

To gain confidence that the extension to CPM+OO remained sound, we followed a similar testing approach to that described in Section 4, but using a wider selection of the example SCOOP programs distributed with EiffelStudio (since the model is now expressive enough to simulate them). In addition, we also: *(i)* used a number of simple sequential programs (i.e. SCOOP programs with only one processor) to focus some testing on the new rules for non-separate calls; and *(ii)* performed "regression testing", in the sense of ensuring that CoreSCOOP programs do not behave differently in CPM+OO to basic CPM.

CPM+OO does not yet cover all of SCOOP: many of the Eiffel-specific mechanisms (agents, once routines, exceptions) and their interactions with SCOOP have not been captured, nor have some advanced run-time mechanisms such as separate callbacks. We also have ignored inheritance for the moment (following [19]), viewing it as an advanced typing mechanism and a separate problem for us to tackle. We do not anticipate substantial difficulty in adding them to CPM+OO; it is our plan to eventually include them by the same methodology, which we view as a promising, practical means of facilitating verification for a rich, complex concurrency model like SCOOP.

Run-Time Alternatives and Approximations An alternative to this gradual extension of CPM is to use it as a basis for exploring and prototyping *alternative semantics*. For SCOOP this is particularly important, since the model has so many competing semantics; most recently a proposal to replace each FIFO queue with a queue of queues [29]. Changing the GTS implementation of the waiting queue, for example, is relatively straightforward due to the model's modularity (see Section 3). Rather than changing the SCOOP compiler first, and risking the discovery of fundamental problems after having committed the time, we propose modifying CPM first, comparing execution traces, and ensuring that the changes retain the high-level guarantees of the model and any other desired properties. We are exploring this usage of CPM as ongoing work, but envisage that such prototyping can be achieved in an analogous way to adding object support: modify the type graph first (e.g. replace the FIFO queue with a queue of queues), revise the affected rules, and test.

A similar idea is to implement *approximations* of CPM directly in the GTS, by plugging in different scheduling rules. As an example, we replaced the FIFO queues of processors with bags (see [26]), thereby removing the guarantee of processors executing their requests in the order they were received.

This is an over-approximation in the sense that all the behaviours of FIFO queues are included in the state space, but several other infeasible behaviours are included too (hence verification of the approximation implies verification of the program, but a counterexample may simply be spurious). Going further, we could, for example, over-approximate CPM by a Petri net (also represented as a GTS).

Translating SCOOP Programs to GROOVE We are developing a tool (to be published as part of a Master's thesis) that automatically translates SCOOP programs to input graphs for GROOVE. The tool targets the same subset of the SCOOP language that CPM+OO prototype formalises (it completely handles, for example, all of the SCOOP programs in [26]). The current prototype first creates a syntax tree of the input classes using the ANTLR4 parser generator in conjunction with an existing SCOOP grammar. Since the input graph requires some typing information (for example, there are different action nodes for integer and reference assignment), the tool passes through the syntax tree twice; first to gather typing information, and then again to generate an intermediate representation of the program that is closely related to the input graph. Finally, the tool passes through the intermediate representation and generates the corresponding input graph as an XML file conforming to the Graph eXchange Language. This graph can then be interpreted and analysed by GROOVE.

6 Verification of SCOOP Programs

In this section we explore how a SCOOP program, once translated to our run-time model in GROOVE, can be verified by (bounded) model checking. After discussing the kinds of properties that can be checked, we illustrate the detection of deadlock in a faulty Dining Philosophers SCOOP solution, and obtain some first verification impressions in a small evaluation of five SCOOP programs.

Verification The GROOVE model checker can be used for automatic analyses that are based on the idea of determining the presence (or absence) of a state that violates some expected property of the program. One such property—the absence of deadlocks—provided the initial motivation for this work. The range of properties that can be verified, however, is much broader; two contrasting examples include the absence of calls to void (null) object references, and the absence of states that violate postconditions (see [26]). This is achieved by extending the run-time model with a set of error rules (assigned the highest priority) that match if and only if the current configuration violates a particular property. An error rule for deadlock, for example, will match if there is a cycle of processors in states preceding lock actions, such that each lock action requires, in turn, a resource held by the next in the cycle. To catch a void call, on the other hand, an error rule will match if a processor is in a state immediately before an action that targets a void reference variable. Then, verification by model checking is simply a matter of expressing—in a temporal logic formula over rules—that none of these error rules are ever applied in the state space. For programs that have an infinite state space (our examples here do not, but those derived from general SCOOP programs may), GROOVE supports *bounded* model checking, which, although unable to fully guarantee correctness, does provide a means of searching for the presence of counterexamples. See [7] for details on bounded model checking with GROOVE.

Recall the Dining Philosophers example from Listing 1. This implementation avoids the possibility of deadlock because the `eat` method requires the simultaneous acquisition of locks on the handlers of the forks (in CPM, this implicit locking is expressed in a single action). Suppose that the philosophers instead call `bad_eat`, as given in Listing 2. This implementation permits executions that lead to deadlock, since philosophers now pick up their forks in turn (which in CPM, then maps to two distinct locking actions).

Listing 2: Snippet of a Dining Philosophers implementation that may deadlock

```
1  bad_eat
2    do
3      print ("Philosopher " + Current.id.out  + " waiting for left fork.%N")
4      pickup_left_then_right (left_fork)
5    end
6  pickup_left_then_right (left: separate FORK)
7    do
8      print ("Philosopher " + Current.id.out  + " waiting for right fork.%N")
9      pickup_right (right_fork)
10   end
11 pickup_right (right: separate FORK)
12   do
13     -- Here, eating takes place
14   end
```

In particular, if every philosopher locks the handlers of their left forks by reaching line 8, the system will deadlock since every fork is locked, preventing the philosophers from entering `pickup_right`. Using the error rules for deadlock and the model checker of GROOVE, faulty executions are automatically unearthed and reported, i.e. paths through the state space from the initial configuration to states exhibiting the structural "deadlock pattern". The relevant part of such a deadlocked state for two philosophers is given in Figure 5. Here, the philosophers have already locked their left forks, and both are waiting to lock their right ones (on reference variables `r_2`). Since the right fork of each philosopher is the already-locked left fork of the other, neither processor can fire the action, and the system is deadlocked.

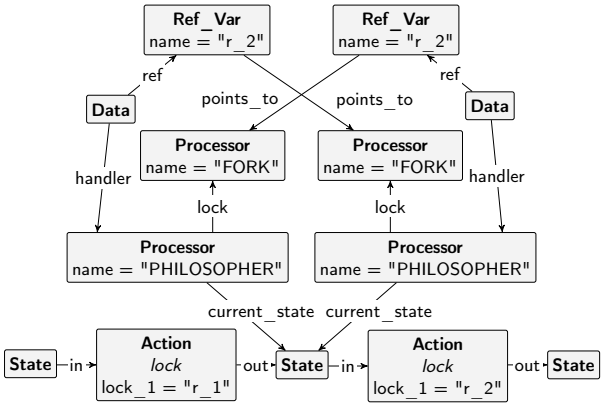

Figure 5: A deadlocked CPM configuration

While this particular bug is somewhat contrived, it does illustrate a discord between the programmer's level of abstraction—"here are the concurrent objects that my method needs"—and a run-time that attempts to handle it all under the hood, but can ultimately fail if the programmer ignores (or is unaware of) how it works. Beyond this example, there are more subtle ways in which deadlock can unintentionally and accidentally be introduced in SCOOP programs [28].

Evaluation To obtain some initial impressions of verification performance, we ran a small study on the current CPM+OO prototype. We devised ten benchmarks from various configurations of five SCOOP programs: Dining Philosophers, both a version that calls `eat` (DPE) and another that calls `bad_eat` (DPB), single-element Producer Consumer (PC), Dining Savages (DS), and Cigarette Smokers (CS). These are available at [26] and were adapted (i.e. to replace unsupported features like inheritance) from the example SCOOP programs provided with EiffelStudio (except for CS, which we implemented).

First, we used the current version of our translation tool to generate input graphs. Then, for these programs, we recorded data about graph sizes, the time to check for deadlock, the time to explore the full (finite) state spaces, and peak memory usage. Table 1 presents this information, computed on an off-the-shelf workstation with Intel Core i7-4810MQ CPU and 16 GB main memory. The presented values are the medians of five tests. Here, n refers to the number of philosophers (for DPE and DPB), the number of elements to produce (for PC), or, for DS, respectively the pot size, number of

Table 1: First impression of verification performance

Program (n)	Start Graph	Final Graph	LTL Deadlock		Full state space		
	(nodes/edges)	(nodes/edges)	time (s)	(states/transitions)	time (s)	(states/transitions)	Mem. [stddev] (GB)
DPE (2)	326/496	362/582	1.10	824/838	1.18	824/838	0.65 [0.11]
DPE (5)	326/496	389/653	32.37	20,428/20,906	29.10	20,428/20,906	4.23 [0.98]
DPB (2)	322/488	378/644	0.84	708/712	1.33	1,108/1,134	0.55 [0.09]
DPB (5)	322/488	447/836	175	74,942/77,378	204	122,714/127,425	5.62 [0.20]
PC (5)	371/563	393/621	3.51	2,152/2,194	3.30	2,152/2,194	0.64 [0.14]
PC (20)	371/563	393/621	12.98	8,362/8,539	12.84	8,362/8,539	1.42 [0.24]
DS (2,2,2)	440/668	470/749	11.48	5,976/6,081	10.82	5,976/6,081	1.41 [0.32]
DS (2,3,2)	441/668	478/769	388	103,190/106,260	256	103,190/106,260	5.71 [0.29]
DS (2,4,1)	441/668	486/789	2,448	396,011/414,462	941	306,401/319,018	7.28 [0.63]
CS	559/866	608/1000	417	65,275/70,008	370	65,275/70,008	5.27 [0.23]

savages, and hunger. Time was measured using System.nanoTime(); the results represent the elapsed (wall clock) time in seconds. Memory usage was measured using Java's MemoryPoolMXBean and related classes. Finally, the exploration strategies used were `bfs`/`final`/`infinite` and `ltl`(`prop` = `!F error_deadlock`)/`final`/`infinite` for the full state space and LTL exploration respectively.

The graph sizes differ little between initial and final states, with the only variation due to the creation and manipulation of processors, their waiting queues, and objects in the local memory. Note the performance discrepancy between LTL checking and full state space exploration. For DPB, which may deadlock, LTL checking is faster since finding one counterexample is enough to return an answer. For all the other programs, which do not deadlock, checking the formula incurs an overhead. Across most of the benchmarks, we would argue that the times are acceptable and practical (especially given the infeasibility of model checking the Maude semantics [19]). An exception is DS, where the overhead for LTL checking is substantial for $n = (2,4,1)$. Understanding the reasons for this is part of an ongoing, broader investigation into the scalability and limits of the tool for verifying SCOOP programs.

7 Conclusion

Related Approaches for GTS-based Specification and Analysis of Concurrent OO Programs Verifying concurrent object-oriented programs with GTS-based models is an emerging trend in software specification and analysis, especially for approaches rooted in practice. See [23] for a good overview discussion, based on a lot of personal experience, on the general appropriateness of GTS for this task.

Closest to our semantic run-time model is the QDAS model presented in [12], which represents an asynchronous, concurrent waiting queue based model with *global memory* as GTS, for verifying programs written in Grand Central Dispatch. Despite the formalisation as GTS, there is, however, no direct compiler to GTS yet. The Creol model of [17] focuses on asynchronous concurrent models but without more advanced remote calls via queues as needed for SCOOP. Analysis of the model can be done via an implementation in a term rewriting tool [16]. Existing GTS-based models for Java only translate the code to a typed graph similar to the control-flow sub-graph of CPM [24, 6]. A different approach is taken by [9], which abstracts a GTS-based model for concurrent OO systems [10] to a finite state model that can be verified using the SPIN model checker. GROOVE itself was already used for verifying concurrent distributed algorithms on an abstract GTS level [13], but not on a run-time level as in our approach. However, despite the intention to apply generic frameworks for the specification, analysis, and verification of object-oriented concurrent programs, e.g. in [30, 8], there are no existing publicly available tools implementing this long-term goal that are powerful enough for SCOOP.

Outlook Our current approach allows for automatically verifying SCOOP programs, with the help of a simple toolchain consisting of a compiler from SCOOP to a GTS-based run-time model that then can be analysed and verified with GROOVE. A streamlined instance of our toolchain, including a publicly available version of the compiler, will be available soon at [26]. As already mentioned, our run-time model can be seen as another operational semantics for SCOOP programs: a more detailed formal comparison with competing formalisations, e.g. [19], is currently on the way based on a more stringent formalisation of the CPM model and its extensions.

The given verification approach and modelling choices can also be applied to other concurrent asynchronous libraries and languages, e.g. the alternative concurrent Eiffel model CAMEO [3], and the existing GTS formalisation of Grand Central Dispatch [12]. As a future step, we want to include verification approaches beyond the strategies of GROOVE, which will depend on novel abstraction techniques for CPM (and its extensions), e.g. in the spirit of pattern abstraction [25], or cluster abstraction [1]. As a lot of verification properties can be boiled down to MSO properties on the underlying GTS, we also plan to enrich the verification techniques for concurrent asynchronous object-oriented programs with ideas from program logics for GTS, e.g. as detailed in [15, 22]. We also plan to publish the current toolchain in a more convenient front end by providing a bridge from existing SCOOP IDEs to GROOVE.

Funding/Acknowledgements. The underlying research was partially funded by the European Research Council under FP7/2007-2013 / ERC Grant agreement no. 291389. We thank B. Meyer and the SCOOP community for valuable "scooped" discussions and A. Rensink for feedback on the internals of GROOVE. Finally, we thank the anonymous GaM referees for their insightful comments, which helped to improve this paper.

References

[1] P. Backes & J. Reineke (2014): *Analysis of Infinite-State Graph Transformation Systems by Cluster Abstraction*. In: *Proc. VMCAI 2015*, *LNCS* 8931, Springer, pp. 135–152, doi:10.1007/978-3-662-46081-8_8.

[2] P. Baldan, A. Corradini & B. König (2008): *Unfolding Graph Transformation Systems: Theory and Applications to Verification*. In: *Concurrency, Graphs and Models*, *LNCS* 5065, Springer, pp. 16–36, doi:10.1007/978-3-540-68679-8_3.

[3] P. J. Brooke & R. F. Paige (2009): *Cameo: an alternative model of concurrency for Eiffel*. *Formal Aspects of Computing* 21(4), pp. 363–391, doi:10.1007/s00165-008-0096-1.

[4] P. J. Brooke, R. F. Paige & J. L. Jacob (2007): *A CSP model of Eiffel's SCOOP*. *Formal Aspects of Computing* 19(4), pp. 487–512, doi:10.1007/s00165-007-0033-8.

[5] G. Caltais & B. Meyer (2014): *Coffman Deadlocks in SCOOP*. In: *Proc. NWPT 2014*. Online version at `http://arxiv.org/abs/1409.7514`.

[6] A. Corradini, F. L. Dotti, L. Foss & L. Ribeiro (2004): *Translating Java Code to Graph Transformation Systems*. In: *Proc. ICGT 2004*, *LNCS* 3256, Springer, pp. 383–398, doi:10.1007/978-3-540-30203-2_27.

[7] G. Delzanno & R. Traverso (2013): *Specification and Validation of Link Reversal Routing via Graph Transformations*. In: *Proc. SPIN 2013*, *LNCS* 7976, Springer, pp. 160–177, doi:10.1007/978-3-642-39176-7_11.

[8] F. L. Dotti, L. M. Duarte, L. Foss, L. Ribeiro, D. Russi & O. Marchi dos Santos (2005): *An Environment for the Development of Concurrent Object-Based Applications*. In: *Proc. GraBaTs 2004*, *ENTCS* 127, Elsevier, pp. 3–13, doi:10.1016/j.entcs.2004.12.026.

[9] A. P. Lüdtke Ferreira, L. Foss & L. Ribeiro (2007): *Formal Verification of Object-Oriented Graph Grammars Specifications*. In: *Proc. GT-VC 2006*, *ENTCS* 175, Elsevier, pp. 101–114, doi:10.1016/j.entcs.2007.04.020.

[10] A. P. Lüdtke Ferreira & L. Ribeiro (2005): *A Graph-based Semantics For Object-oriented Programming Constructs*. In: *Proc. CTCS 2004*, *ENTCS* 122, Elsevier, pp. 89–104, doi:10.1016/j.entcs.2004.06.053.

[11] *Grand Central Dispatch (GCD) Reference*. `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`. Acc.: Dec. 2014.

[12] G. Geeraerts, A. Heußner & J.-F. Raskin (2013): *Queue-Dispatch Asynchronous Systems*. In: *Proc. ACSD 2013*, IEEE, pp. 150–159, doi:10.1109/ACSD.2013.18.

[13] A. H. Ghamarian, M. de Mol, A. Rensink, E. Zambon & M. Zimakova (2012): *Modelling and analysis using GROOVE*. *Intern. J. on Softw. Tools for Techn. Trans.* 14(1), pp. 15–40, doi:10.1007/s10009-011-0186-x.

[14] *GRaphs for Object-Oriented VErification GROOVE (project web page)*. `http://groove.cs.utwente.nl/`. Acc.: Dec. 2014.

[15] A. Habel & K.-H. Pennemann (2009): *Correctness of high-level transformation systems relative to nested conditions*. *Mathem. Struct. in Comp. Sci.* 19(2), pp. 245–296, doi:10.1017/S0960129508007202.

[16] E. B. Johnsen, O. Owe & E. W. Axelsen (2005): *A Run-Time Environment for Concurrent Objects With Asynchronous Method Calls*. In: *Proc. WRLA 2004*, *ENTCS* 117, Elsevier, pp. 375–392, doi:10.1016/j.entcs.2004.06.012.

[17] E. B. Johnsen, O. Owe & I. Chieh Yu (2006): *Creol: A type-safe object-oriented model for distributed concurrent systems*. *Theor. Comput. Sci.* 365(1-2), pp. 23–66, doi:10.1016/j.tcs.2006.07.031.

[18] H. Kastenberg & A. Rensink (2006): *Model Checking Dynamic States in GROOVE*. In: *Proc. SPIN 2006*, *LNCS* 3925, Springer, pp. 299–305, doi:10.1007/11691617_19.

[19] B. Morandi, M. Schill, S. Nanz & B. Meyer (2013): *Prototyping a Concurrency Model*. In: *Proc. ACSD 2013*, IEEE, pp. 170–179, doi:10.1109/ACSD.2013.21.

[20] P. Nienaltowski (2007): *Practical framework for contract-based concurrent object-oriented programming*. Doctoral dissertation, ETH Zürich.

[21] J. S. Ostroff, F. Ahmadi Torshizi, H. F. Huang & B. Schoeller (2009): *Beyond contracts for concurrency*. *Formal Aspects of Computing* 21(4), pp. 319–346, doi:10.1007/s00165-008-0073-8.

[22] C. M. Poskitt & D. Plump (2014): *Verifying Monadic Second-Order Properties of Graph Programs*. In: *Proc. ICGT 2014*, *LNCS* 8571, Springer, pp. 33–48, doi:10.1007/978-3-319-09108-2_3.

[23] A. Rensink (2010): *The Edge of Graph Transformation - Graphs for Behavioural Specification*. In: *Graph Transformations and Model-Driven Engineering*, *LNCS* 5765, Springer, pp. 6–32, doi:10.1007/978-3-642-17322-6_2.

[24] A. Rensink & E. Zambon (2009): *A Type Graph Model for Java Programs*. In: *Proc. FMOODS 2009*, *LNCS* 5522, Springer, pp. 237–242, doi:10.1007/978-3-642-02138-1_18.

[25] A. Rensink & E. Zambon (2012): *Pattern-Based Graph Abstraction*. In: *Proc. ICGT 2012*, *LNCS* 7562, Springer, pp. 66–80, doi:10.1007/978-3-642-33654-6_5.

[26] *Supplementary Material*. `http://www.swt-bamberg.de/gam2015_supp/`.

[27] F. A. Torshizi, J. S. Ostroff, R. F. Paige & M. Chechik (2009): *The SCOOP Concurrency Model in Java-like Languages*. In: *Proc. CPA 2009*, *Concurrent Systems Engineering Series* 67, IOS Press, pp. 7–27, doi:10.3233/978-1-60750-065-0-7.

[28] S. West, S. Nanz & B. Meyer (2010): *A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model*. In: *Proc. ICFEM 2010*, *LNCS* 6447, Springer, pp. 597–612, doi:10.1007/978-3-642-16901-4_39.

[29] S. West, S. Nanz & B. Meyer (2015): *Efficient and reasonable object-oriented concurrency*. In: *Proc. PPoPP 2015*, ACM, pp. 273–274, doi:10.1145/2688500.2688545.

[30] E. Zambon & A. Rensink (2011): *Using Graph Transformations and Graph Abstractions for Software Verification*. In: *Proc. ICGT-DS 2010*, *ECEASST* 38.

[31] E. Zambon & A. Rensink (2014): *Solving the N-Queens Problem with GROOVE - Towards a Compendium of Best Practices*. In: *Proc. GT-VMT 2014*, *ECEASST* 67.